Universidad Complutense de Madrid

Problema de las Monedas

Métodos Algorítmicos En Resolución De Problemas

Facultad de Informática

Beatriz Herguedas Pinedo

Madrid Junio 2019

The first 90 percent of the code accounts for the
first 90 percent of the development time...
The remaining 10 percent of the code accounts
for the other 90 percent of the development time.

Ninety-ninety rule.

Abstract

En el presente trabajo se realizará la prueba exhaustiva de diferentes implementaciones algorítmicas para la resolución del problema de las monedas (también conocido como problema del cambio). Para ello nos centraremos en los principales algoritmos empleados para obtener soluciones de este problema: estrategia voraz, ramificación y poda, programación dinámica y vuelta atrás. Finalmente se compararán y contrastarán los resultados de los diferentes métodos utilizados, centrándonos principalmente en el coste de resolver un problema semejante y su eficiencia.

Índice general

1. Introducción **1**

2. Aclaraciones iniciales **3**

3. Método voraz **7**
3.1. Implementación 7
3.2. Pruebas 8
3.2.1. Primera prueba 9
3.2.2. Segunda prueba 10

4. Método de programación dinámica **13**
4.1. Implementación 13
4.1.1. Con matriz 13
4.1.2. Con vector 15
4.2. Pruebas 15

5. Método de vuelta atrás **17**
5.1. Implementación 17
5.2. Prueba 19

6. Método de ramificación y poda **21**
6.1. Implementación 22
6.1.1. Cota pesimista 22
6.1.2. Cota optimista 22
6.2. Pruebas 23

7. Análisis y comparación **25**
7.1. Programación dinámica: matriz vs vector 25
7.2. Ramificación y poda: poda mínima vs poda óptima 27
7.3. Voraz vs programación dinámica 28
7.4. Voraz vs ramificación y poda 29
7.5. Voraz vs vuelta atrás 30
7.6. Programación dinámica vs ramificación y poda 31
7.7. Programación dinámica vs vuelta atrás 32
7.8. Ramificación y poda vs vuelta atrás 33

8. Conclusiones **35**

Bibliografía **37**

Capítulo 1

Introducción

En la literatura científica podemos encontrar diferentes dilemas relacionados con los sistemas monetarios. En este estudio nos centraremos en el problema del cambio de monedas: dado un sistema monetario[1] se busca determinar la forma de pagar una determinada cantidad utilizando el menor número de monedas posible.

Un **sistema monetario** puede expresarse como un conjunto finito $M = \{m_0, m_1, ..., m_n\}$ de valores enteros positivos distintos, ordenados de forma creciente.

Si planteamos el problema de forma que dada una cantidad entera positiva C, queremos encontrar un conjunto de números enteros no negativos $\{w_0, w_1, ..., w_n\}$, donde cada incógnita w_i representa cuántas monedas de valor m_i se utilizan, tales que minimicen el número total de monedas que se utilizan para sumar C. Podemos expresar el problema formalmente, buscando minimizar la siguiente cantidad :

$$\sum_{i=1}^{n} w_i$$

[1] Un sistema monetario es un conjunto formado por distintos tipos de monedas de un cierto valor. Un ejemplo de sistema monetario es el euro, formado por las monedas de uno, dos, cinco, diez, veinte y cincuenta céntimos y uno y dos euros.

Y además cumpliendo la restricción de que la suma total de monedas equivale a la cantidad a pagar:

$$\sum_{i=0}^{j} m_i w_i = C$$

Ese es el planteamiento general del problema, aunque para aplicar determinados algoritmos de resolución es necesario que el problema cumpla un mayor número de hipótesis y restricciones, en las siguientes páginas estudiaremos con mayor detenimiento estos casos. [1] [2]

Capítulo 2

Aclaraciones iniciales

Tanto la implementación de los algoritmos como la de las pruebas se ha llevado a cabo en el lenguaje de programación C++.

Para la realización de todas las pruebas se ha utilizado el siguiente esquema, aunque variando la forma de introducir los tipos de monedas en el vector y el tamaño de la prueba en función del algoritmo que estemos probando (algunos soportan casos más grandes que otros). Además, para poder comparar los resultados debidamente, en todas las pruebas se ha dado el mismo valor a la C [1]. El valor elegido para esto ha sido $C = 2 * N$.

```
void prueba(int N, ofstream &salida) {

        // Generamos la semilla del random en función del tiempo
        srand(time(NULL));

        // Recorremos el vector insertando los tipos de monedas
        for (int i = 1; i < N; i++)
                tiposMonedas.push_back(valorMoneda);
```

[1] El valor elegido es arbitrario y se mantiene constante para preservar la coherencia entre los resultados de los algoritmos analizados. Un estudio más detallado sobre los diferentes tamaños de prueba y el impacto sobre los algoritmos a emplear va más allá de los objetivos de este trabajo.

```
        // Le damos un valor a C, que es la cantidad que
        //queremos alcanzar para resolver el problema
        int C = 2 * N;

        // Nos guardamos el tiempo justo antes de realizar la prueba
        int t0 = clock();

        // Realizamos la prueba llamando a la función
        //que calcula la solución
        funcionCorrespondiente(tiposMonedas, C, totalutilizadas);

        // Nos guardamos el tiempo justo al terminar la prueba
        int t1 = clock();

        // Calculamos el tiempo que ha durado la prueba
        double tiempo = double(t1 - t0) / CLOCKS_PER_SEC;

        // Ahora escribimos en los archivos y saccamos por pantalla
}
```

El tamaño de la prueba lo aportamos como se expone a continuación:

```
// Constantes que indican el intervalo del
//número de iteraciones del bucle
int const IT_MIN = 50000000;
int const IT_MAX = 60000000;

// De cuánto en cuánto se va iterando el bucle
int const SUMA = 100000;

// Hacemos las pruebas en el intervalo de iteraciones decidido
for (int i = IT_MIN; i < IT_MAX + 1; i = i + SUMA)
        prueba(i, salida, salidaGraficarX, salidaGraficarY);
```

De cara a analizar las gráficas de las pruebas, es importante aclarar que el tiempo está siempre dado en segundos, y el tamaño de la entrada se refiere al tamaño del vector de tipos de monedas. Además, al realizar

las comparativas entre dos algoritmos distintos, siempre hemos utilizado la implementación que daba mejor coste. Por ejemplo, a la hora de comparar el algoritmo de programación dinámica con otros algoritmos, siempre hemos utilizado la implementación con vector, pues da mejores resultados.

Capítulo 3

Método voraz

Para la resolución del problema con método voraz, es necesario que el problema cumpla una serie de hipótesis iniciales bastante fuertes . Dichas hipótesis son [4] [5]:

1. Los tipos de monedas están ordenados en orden estrictamente creciente.
2. La cantidad de monedas disponible de cada tipo es ilimitada.
3. La moneda de valor más bajo es la de valor 1, que siempre está.
4. Cada tipo de moneda es múltiplo del anterior.

O bien cambiar las dos últimas por la siguiente:

5. Los tipos de monedas son todas las potencias entre 0 y N de una cierta base estrictamente mayor que 1.

3.1. Implementación

La estrategia a seguir consiste en recorrer el vector de tipos de monedas (ordenado crecientemente) comenzando por el final, de forma que primero miramos los tipos de monedas más grandes, y nos vamos quedando con todas las más grandes que podemos utilizar hasta llegar a alcanzar la cantidad C.

La función recibe como parámetros: M, que es el vector que contiene los tipos de monedas; C, que es la cantidad que queremos alcanzar; *totalutilizadas*, que es el número total de monedas que utilizamos para resolver el problema; y w, que es un vector de pares que almacena cuántas monedas de cada tipo utilizamos (estos dos últimos son de salida).

```
vector<int> monedasVoraz1(vector<int> const &M, int C,
int &totalutilizadas, vector<pair<int, int>> &w) {
        vector<int> sol(M.size());
        int falta = C;
        int i = M.size() - 1;
        while (falta != 0 && i > -1) {
                sol[i] = falta / M[i];
                totalutilizadas += sol[i];
                if(sol[i] > 0)
                        w.push_back({ sol[i], M[i] });
                falta = falta % M[i];
                i--;
        }
        return sol;
}
```

3.2. Pruebas

Las pruebas del algoritmo se han realizado de dos formas distintas, con el objetivo de obtener resultados pertinentes para el análisis de su eficiencia y optimalidad.

3.2.1. Primera prueba

En la primera prueba hemos creado el vector de tipos de monedas de tal forma que cumpliera las hipótesis de la 1 a la 4, y posteriormente cambiando la 3 y 4 por la 5.

Coste en tiempo. Probar esto es complicado, pues las hipóteis exigen introducir tipos de monedas con un valor muy elevado enseguida, por lo que no nos permite probar casos muy grandes, pues ni el tipo *int* ni el *longlongint* soportan semejantes valores. Esto nos ha llevado a que los tiempos de las pruebas, como más abajo mostramos (figura 3.1), sean 0 o muy próximas a 0 en todas las iteraciones del bucle, por lo que no merece la pena representarlo gráficamente.

Tamaño	Tiempo
15	0
16	0
17	0
18	0
19	0
20	0
21	0
22	0
23	0
24	0
25	0

(a) Hipótesis 1-4

Tamaño	Tiempo
15	0
16	0
17	0
18	0
19	0
20	0
21	0,002
22	0
23	0
24	0
25	0

(b) Hipótesis 1, 2, 5

Figura 3.1: Tiempos Método Voraz

Coste en memoria. Sin embargo, esto no nos ha impedido examinar el coste en memoria del algoritmo. En las gráficas 3.2 y 3.3 que se muestran a continuación, puede observarse que el coste en memoria de dicho algoritmo es bastante bajo.

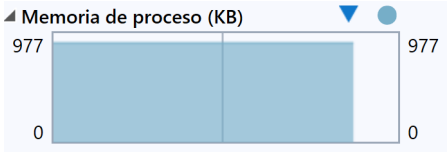


Figura 3.2: Coste en Memoria Hipótesis 1-4

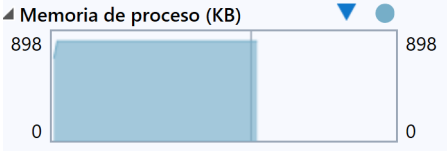


Figura 3.3: Coste en Memoria Hipótesis 1, 2, 5

3.2.2. Segunda prueba

La primera prueba no nos ha permitido probar y analizar el algoritmo como nos gustaría, por lo que hemos realizado una segunda. Esta consiste en introducir números al azar en el vector de tipos de monedas, siempre de forma creciente y con el 1 entre los tipos de monedas, pero sin tener en cuenta las hipótesis 4 y 5. Así logramos analizar el coste en tiempo del algoritmo, aunque sabiendo que el resultado obtenido no será el óptimo, sino una cota superior de este (pues lo que buscamos es un mínimo). De cara a analizar los costes del algoritmo esta prueba nos es muy útil.

Coste en tiempo En la gráfica 3.4 podemos observar que el coste en tiempo del algoritmo voraz es muy pequeño en relación al tamaño de la entrada, pues como puede observarse también en las líneas de tendencia (líneas discontinuas en las gráficas), es un algoritmo que tiene coste lineal, ya que recorre el vector tan solo una vez.

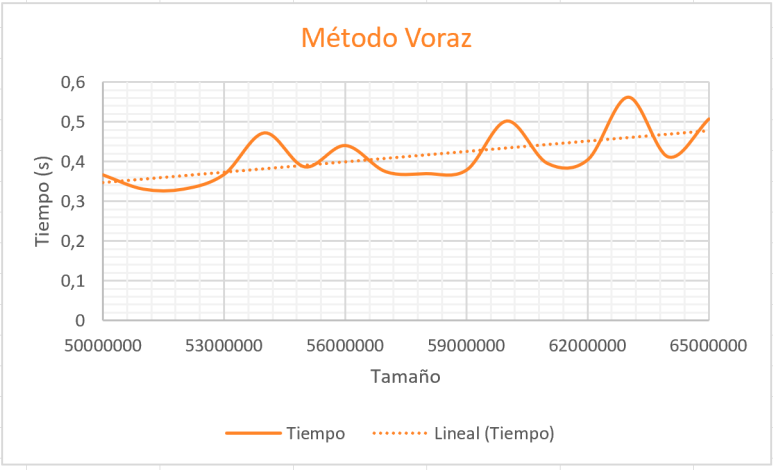


Figura 3.4: Coste en Tiempo Método Voraz

Coste en memoria Analizando casos mucho más grandes que en la primera prueba (hablamos de una entrada rondando varios millones), podemos observar que se utiliza mucha más memoria, pero no llega a ser nada desorbitado (figura 3.5).

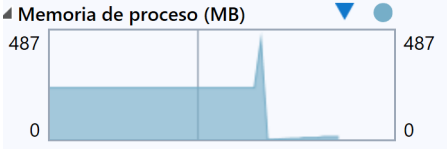


Figura 3.5: Coste en Memoria Método Voraz

Capítulo 4

Método de programación dinámica

De cara a resolver el problema de las monedas con programación dinámica, debemos tener en cuenta la única hipótesis de que la cantidad de monedas disponible de cada tipo sea ilimitada. [8]

Este método puede implementarse de dos formas distintas: almacenando los datos de la recursión en una matriz, o haciéndolo directamente en un vector. La diferencia principal entre las dos implementaciones es el coste en memoria, como veremos más adelante (página 26).

4.1. Implementación

4.1.1. Con matriz

El algoritmo consiste en, partiendo de unos casos base y apoyándonos en los casos anteriores almacenados en la matriz, calcular la solución óptima, utilizando para ello la recursión. [7]

La función recibe como parámetros: v, que es el vector que contiene los tipos de monedas; C, que es la cantidad que queremos alcanzar; y $totalutilizadas$, que es el número total de monedas que utilizamos para

resolver el problema (es un parámetro de salida).

La función devuelve un par con el número mínimo de monedas necesario para resolver el problema y el vector que contiene cuántas monedas del tipo $v[i]$ hemos utilizado para ello.

```
pair<int, vector<int>> monedas(vector<int> const &v,
int C, int &totalutilizados){

        int num = INT_MAX, N = v.size();
        Matriz <int> M(N + 1, C + 1);
        for (int j = 1; j < C + 1; ++j) M[0][j] = INT_MAX;

        for (int i = 1; i < N + 1; i++) {
         for (int j = 1; j < C + 1; j++) {
          if (v[i - 1] > j || M[i][j - v[i - 1]] == INT_MAX)
                M[i][j] = M[i - 1][j];
          else
                M[i][j] = min(M[i - 1][j], M[i][j - v[i - 1]] + 1);
         }
        }
        num = M[N][C];
        // Reconstrucción de la solución
        vector<int> cuantas(N + 1);
        if (num != INT_MAX) {
         int i = N, j = C;
         while (j > 0) {
          if (v[i - 1] < j + 1 && M[i][j] != M[i - 1][j]) {
                cuantas[i]++; totalutilizados++;
                j -= v[i - 1];
          }
          else i--;
         }
        }
        return { num, cuantas };
}
```

4.1.2. Con vector

La implementación con vector es muy similar a la vista anteriormente con matriz. La función recibe y devuelve lo mismo que la anterior, tan solo varía en la forma de abordar la recursión y la reconstrucción de la solución, cambiando la matriz por un vector. Por lo tanto, no vamos a incluir aquí la implementación. Para más información consultar el .cpp.

4.2. Pruebas

Para probar este algoritmo, hemos introducido en el vector de tipos de monedas, números enteros positivos de forma aleatoria, pero sin admitir repetidos (pues suponemos que el número de monedas de cada tipo es ilimitado).

Coste en tiempo En las gráficas 4.1 y 4.2 podemos observar que el coste en tiempo del algoritmo de programación dinámica es $C * N$, por lo que el coste varía en función de la cantidad C que queremos alcanzar. En este caso las pruebas se han hecho con $C = 2 * N$, por lo que el coste es cuadrático (polinómico de grado 2).

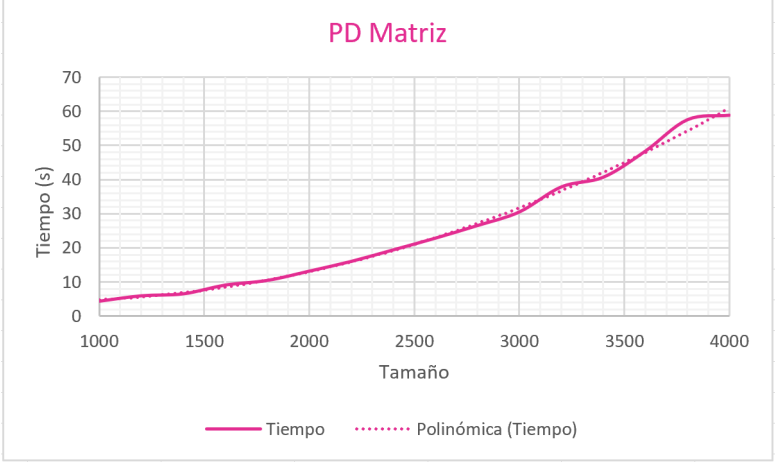


Figura 4.1: Coste en Tiempo Programación Dinámica con Matriz

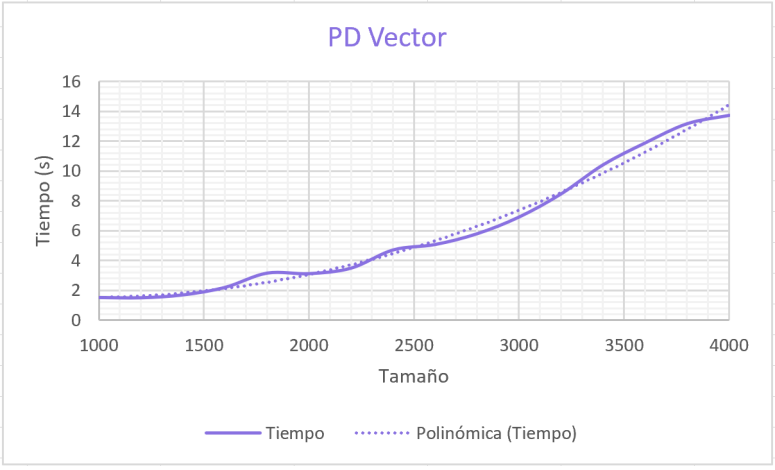


Figura 4.2: Coste en Tiempo Programación Dinámica con Vector

Coste en memoria En las gráficas 4.3 y 4.4, que se muestran a continuación, es destacable la diferencia de coste en memoria entre el algoritmo implementado con matriz y el implementado con vector para casos de prueba con tamaños elevados.

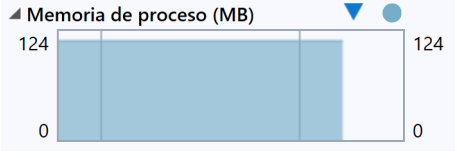


Figura 4.3: Coste en Memoria Programación Dinámica con Matriz

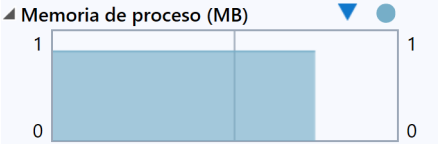


Figura 4.4: Coste en Memoria Programación Dinámica con Vector

Capítulo 5

Método de vuelta atrás

Para la resolución con el método de vuelta atrás, es necesario que el problema cumpla dos hipótesis iniciales:

1. Los tipos de monedas tienen un orden estrictamente decreciente.

2. La cantidad de monedas disponible de cada tipo es ilimitada.

5.1. Implementación

El algoritmo a emplear se basa en la idea de calcular para cada tipo el máximo número de monedas a usar. Por ello, hacemos llamadas recursivas reduciendo el tipo de moneda a considerar, y en cada una probaremos a reducir la cantidad usando desde cero monedas hasta el máximo posible (teniendo en cuenta que esta cantidad está restringuida en función del valor objetivo restante tras cada llamada). Esta aproximación a la resolución del problema evita replicar casos, pues en cada iteración se considerará solamente para cada tipo de moneda el número a usar. Aún así, el coste sigue siendo elevado, pues hay que cubrir todo el espacio de soluciones.

La función recibe como parámetros: M, que es el vector que contiene los tipos de monedas; C, que es la cantidad que queremos alcanzar; *tipo*, que es el tipo de moneda que probamos a usar; $monedasUsadas$ que es un contador de monedas usadas hasta el momento.

```
int vueltaAtras(int C, int tipo, int monedasUsadas,
vector<int> const& M) {
    int  minimo = INT_MAX;

    if (C == 0)
        return monedasUsadas;

    if (tipo < 0)
            return minimo;

    for (int i = 0; i <= C / M[tipo]; i++) {
            if (C - M[tipo] * i >= 0)
                minimo = min(minimo, vueltaAtras(C - M[tipo] * i,
                tipo - 1, i + monedasUsadas, M));
    }
        return minimo;
}
```

5.2. Prueba

Para realizar esta prueba hemos introducido una serie de números aleatorios crecientes en el vector, sin admitir repetidos para cumplir la hipótesis de que el número de monedas de cada tipo sea ilimitado.

Coste en tiempo. En la gráfica 5.1 podemos observar que el coste en tiempo del algoritmo de vuelta atrás, sin realizar ningún tipo de poda, es exponencial (del orden de 2^N). Esto se evidencia con la línea de tendencia de la gráfica.

Figura 5.1: Coste en Tiempo Vuelta Atrás Sin Podas

Coste en memoria En la gráfica 5.2, que se adjunta en más abajo, se puede observar que el algoritmo ocupa escaso espacio de memoria.

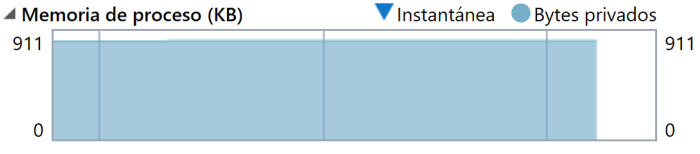


Figura 5.2: Coste en Memoria Vuelta Atrás Sin Podas

Capítulo 6

Método de ramificación y poda

Primero, recoredemos que mantenemos como la hipótesis inicial la posibilidad de escoger monedas ilimitadas de cada tipo.
A su vez, debemos hacer notar que para un sistema monetario que contiene la moneda unidad siempre existe solución para pagar una cantidad C, basta tomar C monedas unidad (esto cumple trivialmente la restricción). Por tanto, podemos aplicar el **esquema optimista-pesimista de ramificación y poda** correctamente.

El nodo a emplear para la implementación del algoritmo se detalla a continuación:

```
struct nodo {
        vector<int> sol; // Array de soluciones parciales
        int k;  // Indica el nivel del árbol explorado
        int cantidad; // Indica la cantidad pagada
        int monedas; // Recuento de monedas utilizadas hasta el momento
        int costeEstimado; // Cota optimista (prioridad)
};
```

6.1. Implementación

Para la implementación del esquema de ramificación y poda se mantuvo una variable durante la ejecución del algoritmo con el *coste mejor* hasta el momento, esta variable al principio equivale a C hasta encontrar la primera solución posible (resultado de tomar todas las monedas unidad); actualizándose su valor además cada vez que encontremos una solución mejor. Es más, en el esquema optimista-pesimista de ramificación y poda podemos ir actualizando esta variable con el coste mejor cada vez que encontremos un nodo factible que puede extenderse a una solución cuyo coste sea menor. Esta extensión se corresponde con la cota superior y nos permite podar más efectivamente a lo largo de la ejecución del algoritmo.

6.1.1. Cota pesimista

La extensión a la cota superior se corresponde con el cálculo de la **cota pesimista**, en este caso hemos optado por tomar la cota pesimista trivial resultante de completar con monedas unidad la cantidad restante por pagar.

```
int calculo_pesimista(int cantidadTotal,
    int cantidadAcumulada, int monedas) {
        return (cantidadTotal - cantidadAcumulada) + monedas;
}
```

6.1.2. Cota optimista

La poda que se realiza toma como cota superior las monedas que llevamos utilizadas más tantas monedas como cantidad nos falte por sumar para alcanzar la cantidad C. Es decir, nos ponemos en el caso peor de que necesitemos monedas de 1 para lo que nos falta por sumar.

```
int calculo_optimista(int cantidadTotal, int cantidadAcumulada,
    int monedas, int monedaMax) {
        return ((cantidadTotal - cantidadAcumulada)
        / monedaMax) + monedas;
}
```

6.2. Pruebas

Como para el algoritmo de ramificación y poda solamente necesita verificar que las monedas van dadas ordenadas en orden creciente y contienen el elemento unitario, se han introducido números al azar en el vector de tipos de monedas para los casos de pruebas (además del elemento unitario).

Coste en tiempo En las gráficas 6.1 y 6.2 podemos observar que el coste en tiempo del algoritmo de ramificación y poda depende mucho del caso en concreto que tenga por entrada y de la poda realizada. No obstante, no deja de ser un caso particular de vuelta atrás con podas que mejoran su tiempo, por lo que, en el caso peor, el coste sería el mismo que el del algoritmo de vuelta atrás: exponencial. La tendencia si puede dislumbrarse en el segundo gráfico, que se ve es a la alza, sin embargo es difícil para estos casos de sistemas monetarios donde abundan las variedades de monedas y las podas son eficaces, encontrar casos de gran coste computacional.

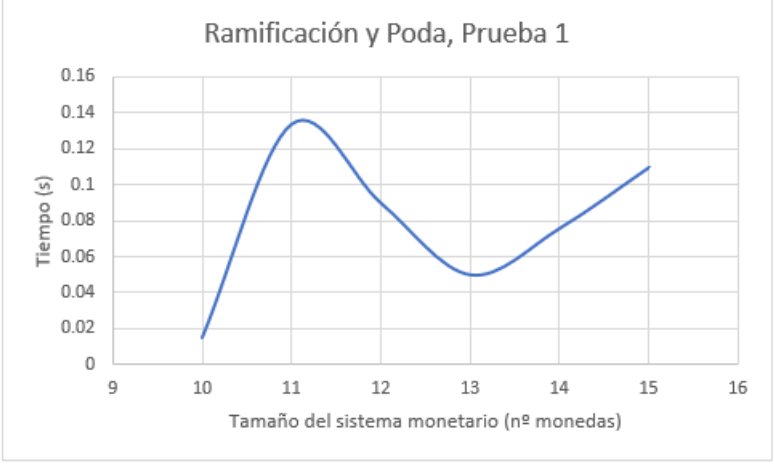


Figura 6.1: Coste en ramificación y poda en un pequeño intervalo de tamaños

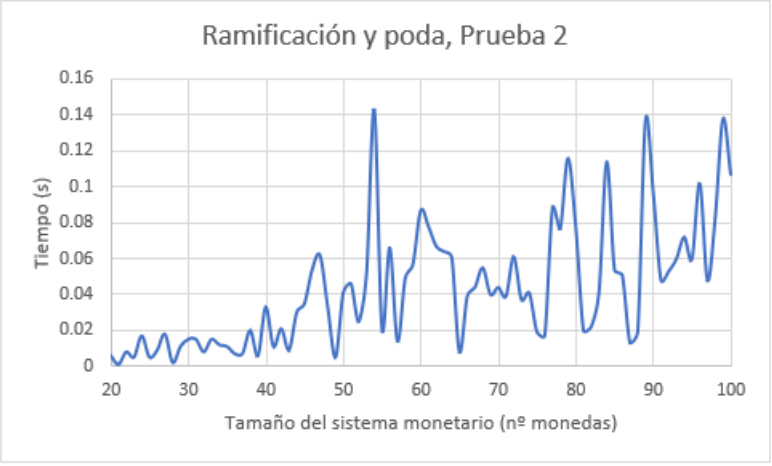


Figura 6.2: Coste en ramificación y poda en un intervalo de tamaños mayor

Coste en memoria En las gráficas 6.3 y 6.4, que se muestran a continuación, se aprecia que este algoritmo no ocupa demasiada memoria.

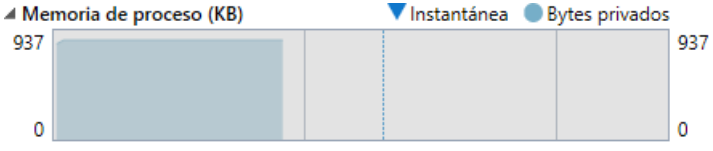


Figura 6.3: Coste en Memoria con intervalo pequeño

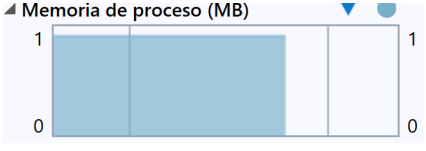


Figura 6.4: Coste en Memoria con intervalo grande

Capítulo 7

Análisis y comparación

En este capítulo vamos a realizar una comparativa entre los costes de cada algoritmo, analizando cada caso con detalle.

7.1. Programación dinámica: matriz vs vector

Coste en tiempo Aunque el coste en tiempo de este algortimo en teoría es el mismo si se implementa con matriz o si se hace con vector, lo cierto es que eso es un coste asintótico, y, por la forma en la que está implementado el código, los accesos a memoria, etc., hay bastante diferencia entre una implementación y otra, siendo mejor la implementación con vector. Esto queda evidenciado en la gráfica 7.1, que se expone a continuación.

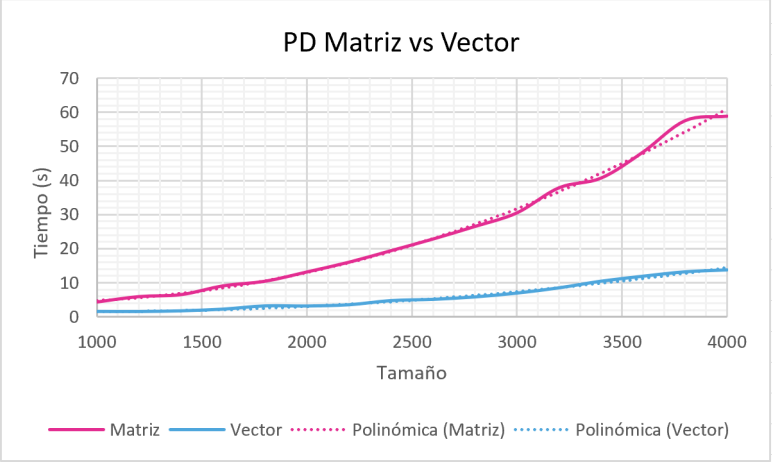
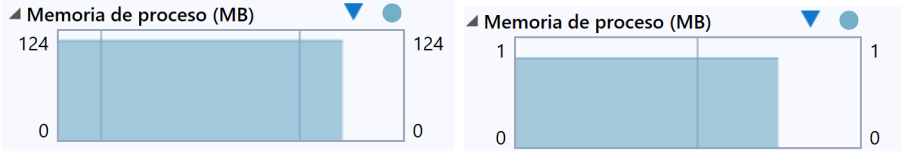


Figura 7.1: Coste en Tiempo Programación Dinámica

Coste en memoria Como puede observarse en la gráfica 7.2, para tamaños entre 1000 y 4000, el coste en memoria del algoritmo implementado con vector es bastante mejor que el coste del algoritmo implementado con matriz. Esto es lógico, pues una matriz ocupa mucho más espacio de memoria que un vector.



(a) Matriz (b) Vector

Figura 7.2: Coste en Memoria Programación Dinámica

7.2. Ramificación y poda: poda mínima vs poda óptima

Coste en tiempo Como es evidente en la figura 7.3, el coste en tiempo del algoritmo con poda óptima es bastante mejor que el coste del algoritmo con poda mínima.

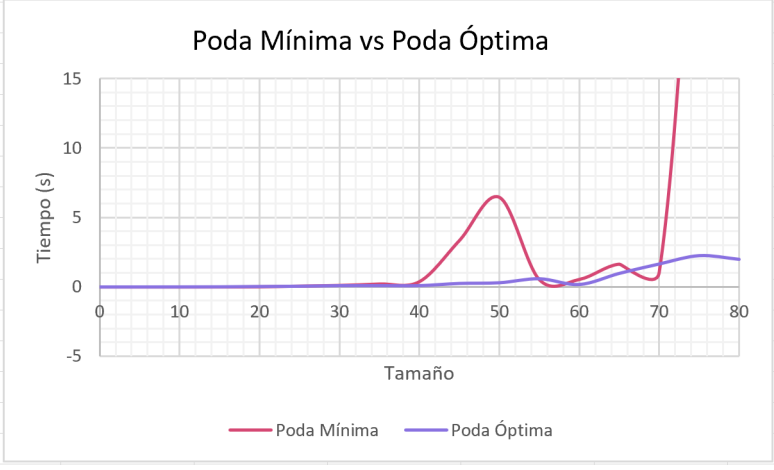
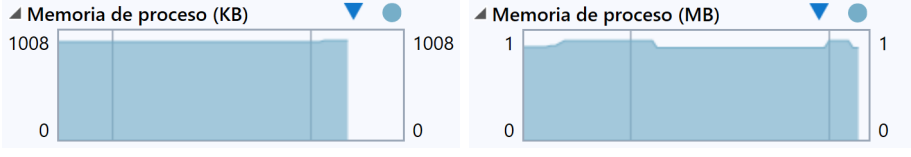


Figura 7.3: Coste en Tiempo Ramificación y Poda

Coste en memoria Como puede observarse en la gráfica 7.4, para tamaños entre 30 y 100, el coste en memoria del algoritmo con poda mínima es ligeramente mejor que el coste del algoritmo con poda óptima.



(a) Poda Mínima (b) Poda Óptima

Figura 7.4: Coste en Memoria Ramificación y Poda

7.3. Voraz vs programación dinámica

Coste en tiempo Es bastante evidente en la gráfica 7.5 que el coste en tiempo del algoritmo voraz es mucho mejor que el coste del algoritmo de programación dinámica (para la prueba hemos tomado la implementación con vector, de cara a comparar los mejores costes de cada algoritmo).

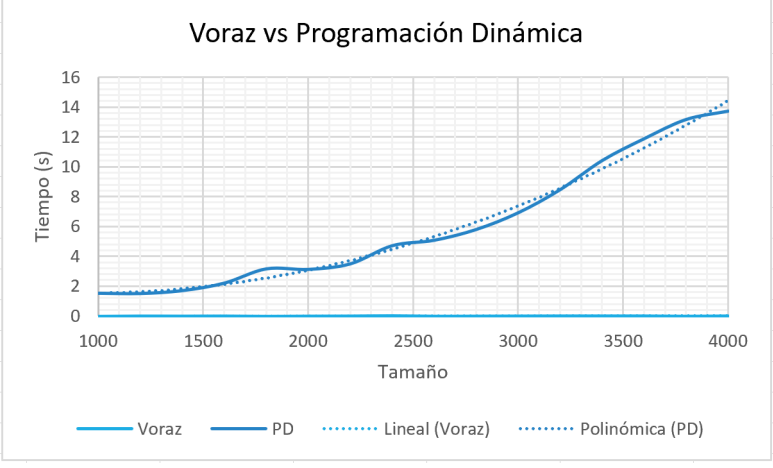
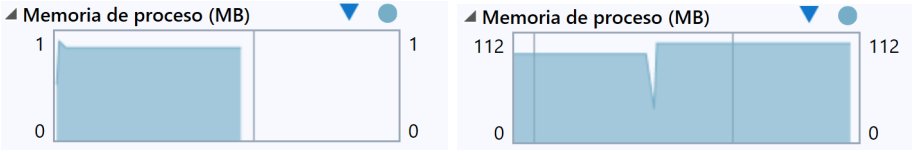


Figura 7.5: Coste en Tiempo Voraz vs PD

Coste en memoria Es evidente por las gráficas de la figura 7.6 que el coste en memoria del algoritmo voraz es mucho mejor que el coste del algoritmo de programación dinámica, medido para entradas entre 1000 y 4000.



(a) Voraz (b) Programación Dinámica

Figura 7.6: Coste en Memoria de Voraz vs PD

7.4. Voraz vs ramificación y poda

Coste en tiempo No es difícil darse cuenta en la figura 7.7 de que el coste en tiempo del algoritmo voraz es también mucho mejor que el coste del algoritmo de ramificación y poda (hemos elegido el óptimo para comparar los mejores tiempos).

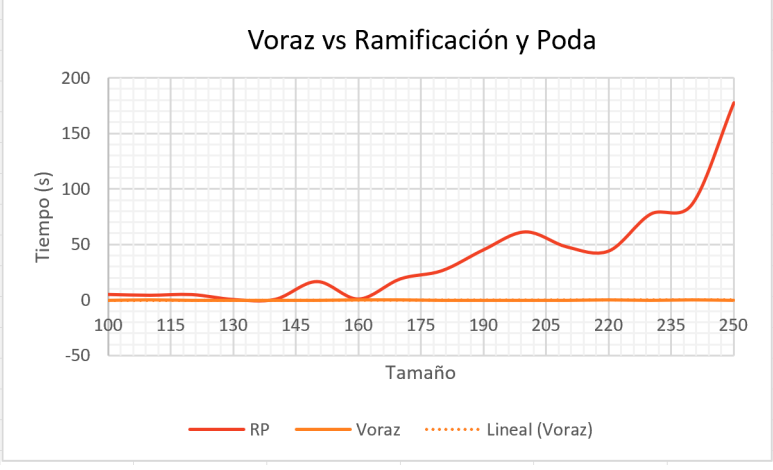
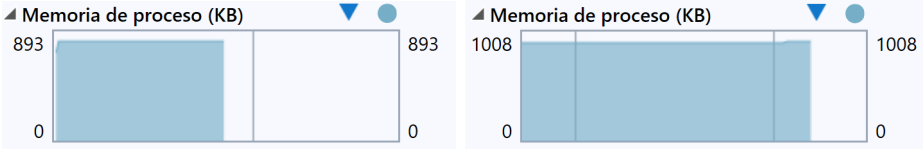


Figura 7.7: Coste en Tiempo de Voraz vs RP

Coste en memoria Como puede observarse en la gráfica 7.8, el coste en memoria del algoritmo voraz es mejor que el coste en memoria del algoritmo de ramificación y poda (mínimo, pues es el que menos memoria utiliza).



(a) Voraz (b) Ramificación y Poda

Figura 7.8: Coste en Memoria Voraz vs RP

7.5. Voraz vs vuelta atrás

Coste en tiempo El coste en tiempo del algoritmo voraz es mejor que el coste del algortimo de vuelta atrás, como queda evidenciado en la gráfica 7.9.

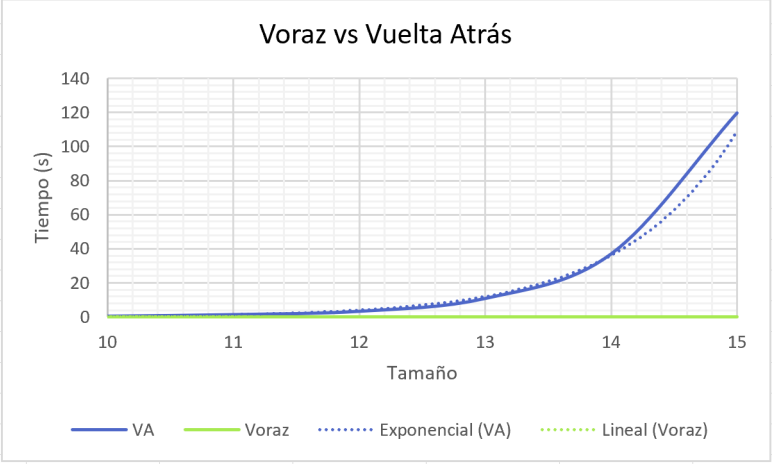
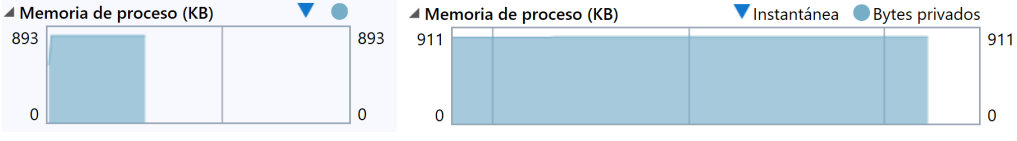


Figura 7.9: Coste en Tiempo Voraz vs VA

Coste en memoria En las gáficas de la figura 7.10 puede observarse que para entradas de tamaños entre 10 y 15, el coste en memoria del algoritmo de vuelta atrás es ligeramente superior al coste del algoritmo voraz.



(a) Voraz (b) Vuelta Atrás

Figura 7.10: Coste en Memoria Voraz vs VA

7.6. Programación dinámica vs ramificación y poda

Coste en tiempo En la siguiente gráfica (7.11), podemos observar que el coste en tiempo del algoritmo de programación dinámica (con vector) es mucho mejor que el coste del algoritmo de ramificación y poda (óptima).

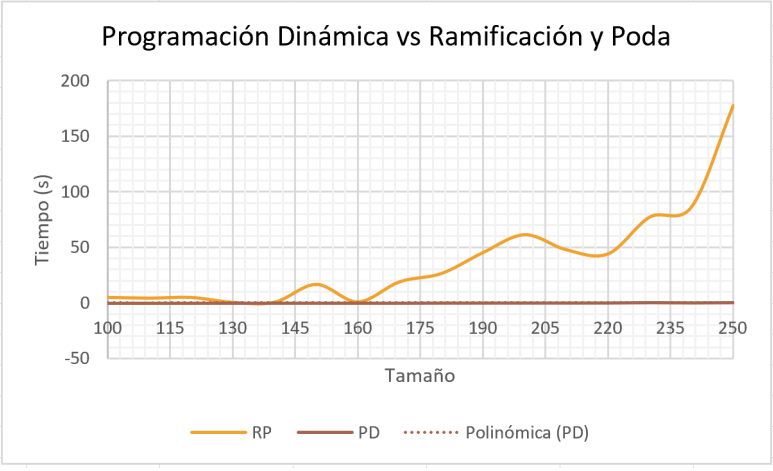
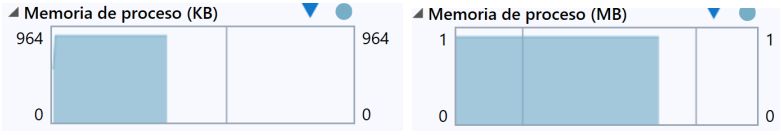


Figura 7.11: Coste en Tiempo PD vs RP

Coste en memoria Para entradas de tamaños entre 100 y 250, el coste en memoria de estos dos algoritmos es muy similar, aunque es ligeramente mejor el coste del algoritmo de programación dinámica (figura 7.12).



(a) Programación Dinámica (b) Ramificación y Poda

Figura 7.12: Coste en Memoria PD vs RP

7.7. Programación dinámica vs vuelta atrás

Coste en tiempo A continuación podemos observar que el coste en tiempo del algoritmo de vuelta atrás es muy superior al coste del algoritmo de programación dinámica (figura 7.13).

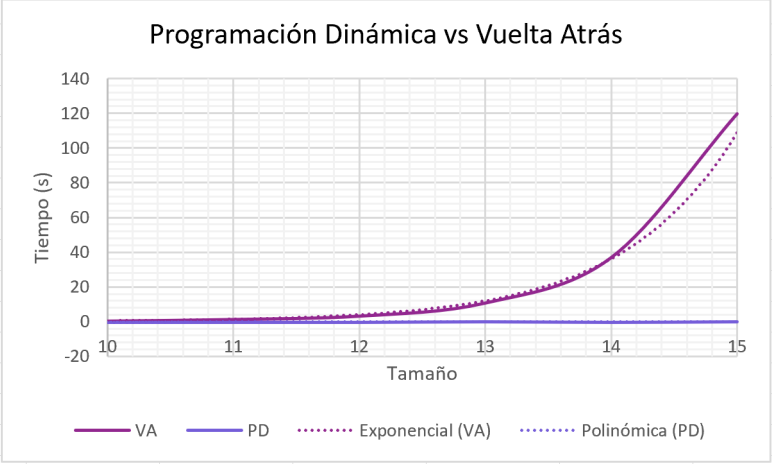
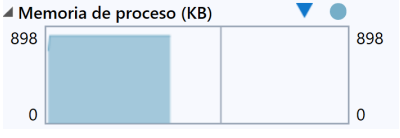
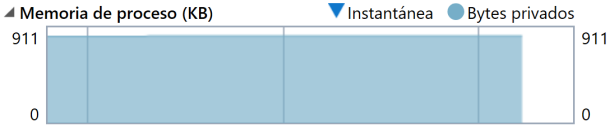


Figura 7.13: Coste en Tiempo PD vs VA

Coste en memoria En las gáficas adjuntas debajo (figura 7.14) puede observarse que para entradas de tamaños entre 10 y 15, el coste en memoria del algoritmo de vuelta atrás es ligeramente superior que el de programación dinámica (con vector).



(a) Programación Dinámica



(b) Vuelta Atrás

Figura 7.14: Coste en Memoria PD vs VA

7.8. Ramificación y poda vs vuelta atrás

Coste en tiempo Y por último, es fácil darse cuenta de que el coste en tiempo del algoritmo de vuelta atrás supera con creces el coste del algoritmo de ramificación y poda (figura 7.15).

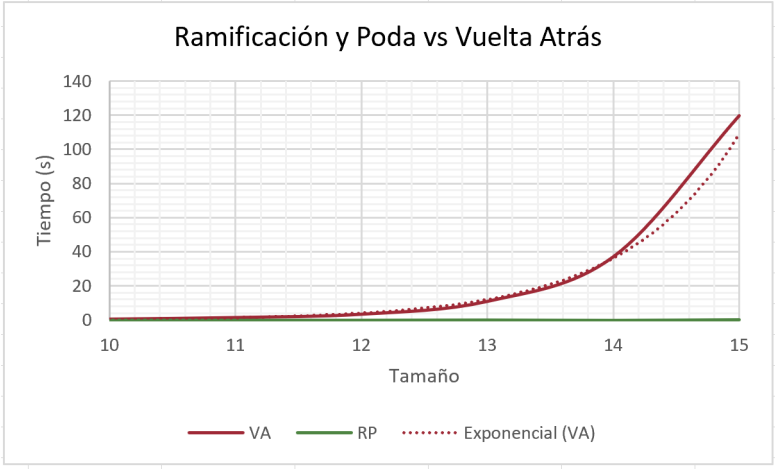
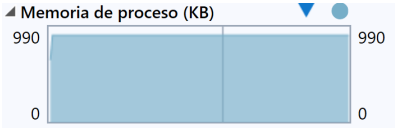
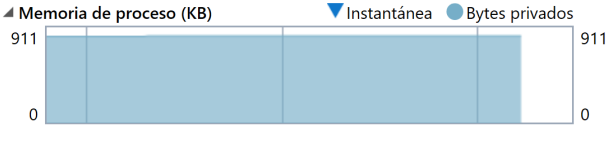


Figura 7.15: Coste en Tiempo RP vs VA

Coste en memoria En las gáficas (figura 7.16) puede observarse que para entradas de tamaños entre 10 y 15, el coste en memoria de ambos algoritmos es muy similar, siendo un pelín superior el del algoritmo de ramificación y poda (mínima).



(a) Ramificación y Poda



(b) Vuelta Atrás

Figura 7.16: Coste en Memoria RP vs VA

Capítulo 8

Conclusiones

Tras las numerosas pruebas exhaustivas realizadas, hemos llegado a la conclusión de que el algoritmo que mejor coste tiene en tiempo es el voraz, pero en contrapunto, es necesario añadir muchas hipóteis al problema para que este algoritmo aporte la solución óptima. A este le sigue el algoritmo de ramificación y poda, aunque su coste en tiempo depende mucho de la poda y la entrada. Depués irá el algoritmo de programación dinámica, y por último el algoritmo de vuelta atrás, pues no realiza ninguna poda, por lo que pasa por todos y cada uno de los nodos del árbol de exploración y esto provoca que tarde mucho tiempo en ello.

En cuanto al coste en memoria, el algoritmo que peor coste tiene es el de programación dinámica implementado con matriz, pues la matriz ocupa mucho espacio en la memoria. Los demás algoritmos andan todos rondando los mismos valores, aunque el coste depende del tamaño de la entrada.

Bibliografía

[1] Adamaszek, Anna y Michal Adamaszek: *Combinatorics of the change-making problem*. Elsevier Ltd, Centre for Discrete Mathematics and its Applications, University of Warwick, Coventry, CV4 7AL, UK, 2009.

[2] Cai, Xuan: *Canonical Coin Systems for Change-Making Problems*. Elsevier Ltd, Ninth International Conference on Hybrid Intelligent Systems, Shenyang, China, 2009.

[3] Chaitanya, Bnsk: *Coinchange Problem*. `https://github.com/bnsk/Coinchange_Problem/blob/master/backtracking_coinchange.c`.

[4] Oliet, Narciso Martí, Yolanda Ortega Mallén y Alberto Verdejo: *Método voraz: pago óptimo con monedas*.

[5] Oliet, Narciso Martí, Yolanda Ortega Mallén y Alberto Verdejo: *Estructuras de Datos y Métodos Algorítmicos*. Ibergaceta Publicaciones, SL, Madrid, segunda edición, 2013.

[6] Ortiz, Angel Luis: *Problema del Cambio*. `https://github.com/rurickturick/Mar/blob/master/Problema%20del%20cambio/Pagar.cpp`.

[7] Verdejo, Alberto: *Matriz.h*.

[8] Verdejo, Alberto: *Programación Dinámica*. Universidad Complutense de Madrid, 2008.

[9] Verdejo, Alberto: *Ramificación y Poda*. Universidad Complutense de Madrid, 2013.